Available online at www.sciencedirect.com

SCIENCE DIRECT

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 127 (2005) 133–151

www.elsevier.com/locate/entcs

Towards a Programming Language for Interaction Nets

Ian Mackie[1]

Department of Computer Science
King's College London
Strand, London WC2R 2LS, UK

Abstract

Interaction nets were introduced almost 15 years ago. Since then they have been put forward as both a graphical programming paradigm and as an intermediate language into which we can compile other languages. Whichever way we use interaction nets, a problem remains in that the language is very primitive. Drawing an analogy with functional programming, we have the λ-calculus but we are missing the functional programming language: syntactic sugar, language constructs, data-structures, etc. The purpose of this paper is to make a first step towards defining such a programming language for interaction nets.

Keywords: Interaction nets, programming language design.

1 Introduction

Interaction nets are particular kinds of graph rewriting systems which have constraints over both the construction of graphs and the corresponding rewrite rules. In fact they are so constrained that it is indeed surprising that they capture all computable functions. The fact that they lend themselves to modelling efficient computation, for instance β-optimal and efficient reduction (see for instance [1,19]) is evidence to that fact that they can play an important rôle in computer science.

There are several implementations of interaction nets in the public domain (for instance [23,17]), which have been developed to demonstrate interaction

[1] Email: ian@dcs.kcl.ac.uk

1571-0661
doi:10.1016/j.entcs.2005.02.015

nets graphically, or to test out various implementation ideas, such as parallelism. These are based on a "pure" calculus of interaction nets (see [5] for such a calculus and details of the operational model), and thus writing programs with it can be understood as analogous to writing functional programs with the pure λ-calculus. What is therefore missing is a substantial *programming language* development: syntax, semantics, implementation, as well as a development environment (tools to facilitate program development, such as graphical previewers). The purpose of this paper is to make a start on the design and definition of such a language, specifically to identify the required features and an appropriate syntax. Interaction nets can be seen as a paradigm of computation, different from functional, logical, or object based.

We see this paper as a first step towards the design and definition of a programming language for interaction nets. Specifically, we address the following issues:

- A syntax for an interaction net programming language, which will provide a notation for writing the basic components of an interaction net system: agents, nets and rules;
- Modularity constructs for the definition of nets and rules which can support code reuse;
- Interaction rule schemes (templates) allowing the definition of families of rules which have the same form;
- Side-effects, including input/output and foreign language interfaces.

We discuss the issues above informally, using examples to motivate our design decisions. We stress that this is work-in-progress, and expect many iterations of some of the features. We also exclude many very important issues in the present paper, for instance type systems, so that we can focus on some of the very basic issues of syntax.

We will draw upon experience gained from languages such as Standard ML [20], Haskell [21], PICT [22], and many others to design and build such a system.

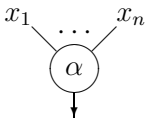
Overview

The rest of this paper is structured as follows. We recall interaction nets in the next section, and explain why they are important. We then, in Section 3, motivate through examples the features that we want in our language. The rest of the paper is then devoted to covering these issues in more detail: Section 4 presents syntax, Section 5 covers modules, Section 6 covers data types, and finally Section 7 covers side effects. We conclude and suggest some future

directions in Section 8.

2 Interaction Nets

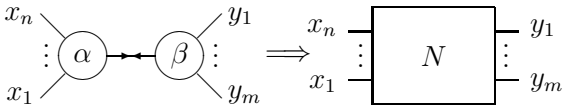
Lafont's Interaction Nets [14] are highly constrained graph rewrite systems. The remarkable feature is that even with these constraints interaction nets can still encode all the computable functions. One of the most beautiful features of interaction nets is that many important properties are obtained directly from their construction, for instance confluence. An interaction net is specified by giving a set Σ of symbols, and a set $\mathcal{R}$ of interaction rules. Each symbol $\alpha \in \Sigma$ has an associated (fixed) *arity*. An occurrence of a symbol $\alpha \in \Sigma$ will be called an *agent*. If the arity of α is n, then the agent has $n+1$ *ports*: a distinguished one called the *principal port* depicted by an arrow, and n *auxiliary ports* labelled $x_1, \ldots, x_n$ corresponding to the arity of the symbol. We represent an agent graphically in the following way:



If $n = 0$ then the agent has no auxiliary ports, but it will always have a principal port.

A net N built on Σ is a graph (not necessarily connected) with agents at the vertices. The edges of the net connect agents together at the ports such that there is only one edge at every port (edges may connect two ports of the same agent). The ports of an agent that are not connected to another agent are called the free ports of the net. There are two special instances of a net that we should point out: a wiring (a net with no agents) and the empty net.

A pair of agents $(\alpha, \beta) \in \Sigma \times \Sigma$ connected together on their principal ports is called an *active pair*, which is the interaction net analogue of a redex. An interaction rule $((\alpha, \beta) \Longrightarrow N) \in \mathcal{R}$ replaces an occurrence of the active pair (α, β) by a net N. The rule has to satisfy a very strong condition: all the free ports are preserved during reduction, and moreover there is at most one rule for each pair of agents. The following diagram illustrates the idea, where N is any net built from Σ.

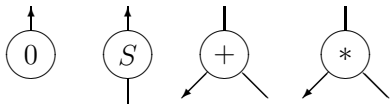


The rewriting process has to create the right-hand side of the net, and make all the connections (re-wirings). Although it may not be the most trivial of operations, it is always a known, constant time operation.

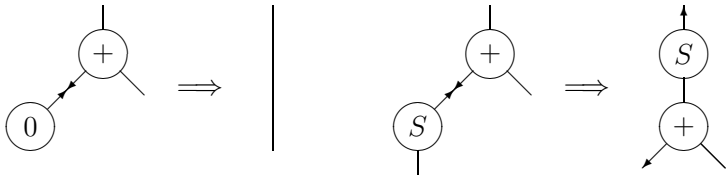
As a direct consequence of the constraints of interaction rules, the system of rewriting is local, and the one-step reduction satisfies the diamond property. An interaction net is said to be in normal form when there are no active pairs, and because of the one-step diamond property, all reductions to normal form are equal up to permutation (if a net normalises, then all reduction sequences normalise to the same net). Of course, there are other, weaker, notions of normal form, such as interface normal form [5], which do not fit this pattern.

Example

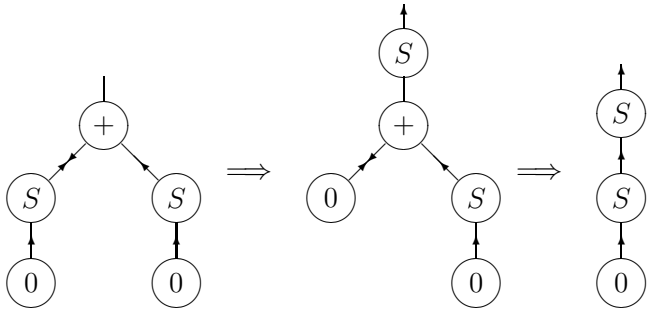
Here we give some very simple examples of interaction nets, which are based on a representation of numbers using interaction nets. First we define agents 0, S, $+$ and $*$:



which allow us to construct nets representing arithmetic expressions. We first introduce two rewrite rules which we leave the reader to relate to the standard equational term rewriting system definition of addition:



A simple net, representing $S(0) + S(0)$ is shown below, where one active pair has been generated. We then show two reductions, which use the previous two rules:

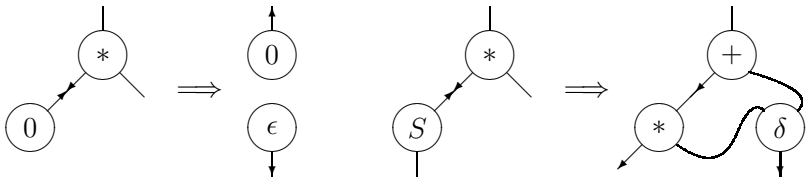


The final net, on the right-hand side, is of course the representation of 2, which is the expected answer.

The next two rules define multiplication. However, these rules are complicated by the fact that they are non-linear. This can be seen by looking at the definition of multiplication as a term rewriting system:

$$*(0, y) \quad \to 0$$
$$*(S(x), y) \to +(*(x, y), y)$$

In the first rule, y is erased, and in the second rule, y is duplicated. When we write interaction rules for this example, we are obliged to preserve the interface, and therefore add extra agents:



The purpose of the agent ϵ is to erase the unused argument, and the purpose of the δ agent is to copy it. We invite the reader to complete this example.

We shall use these examples several times throughout this paper.

Why Interaction Nets

Interaction nets are a generalisation of proof nets for linear logic [9], in a similar way that term rewriting systems are a generalisation of the λ-calculus. Interaction nets are an important model of computation for several reasons:

(i) *All* aspects of a computation are captured by the rewriting rules—no external machinery such as copying a chunk of memory, or a garbage collector, are needed. Interaction nets are amongst the few formalisms which model computation where this is the case, and consequently they can serve as both a low level operational semantics and an object language for compilation, in addition to being well suited as a basis for a high-level programming language.

(ii) Interaction nets naturally capture *sharing*—active pairs can never be duplicated. Thus only normal forms can be duplicated, and this must be done incrementally. Using interaction nets as an *object* language for compilers has offered strong evidence that this sharing will be passed on to the programming language being implemented. One of the most spectacular instances of this is the work by Gonthier, Abadi and Lévy, who gave a system of interaction nets to capture both optimal β-reduction [16] in the λ-calculus [10] (Lamping's algorithm [15]), and optimal reduction for cut-elimination in linear logic [11].

(iii) There is growing evidence that interaction nets can provide a platform for the development of parallel implementations, specifically parallel implementations of sequential programming languages. Using interaction nets as an *object* language for a compiler offers strong evidence that the programming language being implemented may be executed in parallel (*even if there was no parallelism in the original program*). Some strong evidence of this can be found in [23].

The above three points are clear indicators that interaction nets have a rôle to play in computer science every bit as important as the rôles the λ-calculus or term rewriting systems have played over the last few decades. The aim of the present work is to realise some of the potential of interaction nets through the development of a programming language.

3 Programming Language Features

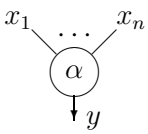
In this section we give an overview of the features that we require our programming language to posses. In particular, we will give our methodology and report on the progress that we have made towards the design of a language for interaction nets. In the following sections we will study each of these features in turn.

3.1 Syntax

The first issue that we have to solve is to devise a notation for writing down interaction nets. We mention two alternative textual languages for interaction nets that have been used in the literature. As as running example, we show the system for numbers and addition used above. Both involve the use of names (we refer the reader to [18] for an alternative name-free version).

Representing Agents

If we have an agent α of arity n:



then we can write this textually as either:

$$\alpha(y, x_1, \ldots, x_n)$$

or

$$y = \alpha(x_1, \ldots, x_n)$$

The first representation simply enumerates all the ports, starting from the principal port. The second takes advantage of the fact that the principal port is unique, in which case we can write what we call a *binding equation*. Note that such a notation would not be so simple if there were more than one principal port (we could not so easily identify a root of the agent). Using the example system of interaction nets from Section 2, we can write the agents as:

$$\mathsf{Z}(x) \qquad \mathsf{S}(x,y) \qquad \mathsf{Add}(x,y,z) \qquad \mathsf{Mult}(x,y,z)$$

using the first notation, and:

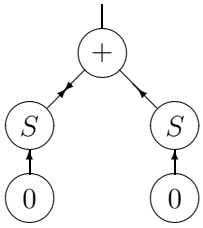
$$x = \mathsf{Z}() \qquad x = \mathsf{S}(y) \qquad x = \mathsf{Add}(y,z) \qquad x = \mathsf{Mult}(y,z)$$

using the second. We can also abbreviate $\mathsf{Z}()$ to just Z when the arity is zero (i.e. when an agent has no auxiliary ports). In both cases the agent Z corresponds to 0, S corresponds to S, Add corresponds to $+$ and Mult corresponds to $*$.

For writing agents, both of these notations seem adequate, and there is no real reason for choosing one notation above the other.

Representing Nets

One of the simplest ways of writing a net would be as a comma separated list of agents. This corresponds to a straightforward flattening of the graph, and of course there are many different ways we could do this depending on the order we choose to enumerate the agents. Using the following net as an example:



we can use the two different representations of agents given previously to generate two different representations of nets as comma separated lists:

$$\mathsf{Add}(x,z,u), \mathsf{S}(x,y), \mathsf{Z}(y), \mathsf{S}(u,v), \mathsf{Z}(v)$$

or

$$x = \mathsf{Add}(z,u), x = \mathsf{S}(y), y = \mathsf{Z}, u = \mathsf{S}(v), v = \mathsf{Z}$$

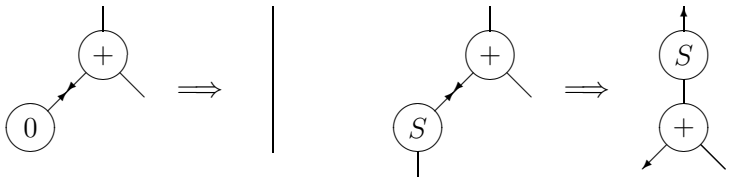
Clearly the first one is more compact, but the second can be simplified by replacing equals for equals, which eliminates some of the names:

$$\mathsf{S}(\mathsf{Z}) = \mathsf{Add}(z, \mathsf{S}(\mathsf{Z}))$$

which is much more compact, and moreover does not require many names.

Representing Rules

Finally we compare the notations for writing interaction rules. The rules for the agent Add in the example system:



can be represented in two different ways again:

$$\mathsf{Add}(a,x,y),\mathsf{Z}(a) \implies x = y$$

$$\mathsf{Add}(a,x,y),\mathsf{S}(a,z) \implies \mathsf{S}(x,b),\mathsf{Add}(z,b,y)$$

or

$$a = \mathsf{Add}(x,y), a = \mathsf{Z} \implies x = y$$

$$a = \mathsf{Add}(x,y), a = \mathsf{S}(z) \implies x = \mathsf{S}(b), z = \mathsf{Add}(b,y)$$

Now the latter two rules can be simplified as before to give:

$$\mathsf{Z} = \mathsf{Add}(x,x) \implies$$

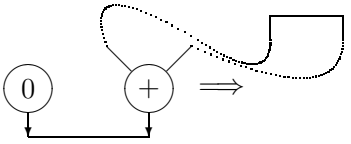
$$\mathsf{S}(\mathsf{Add}(b,y)) = \mathsf{Add}(\mathsf{S}(b),y) \implies$$

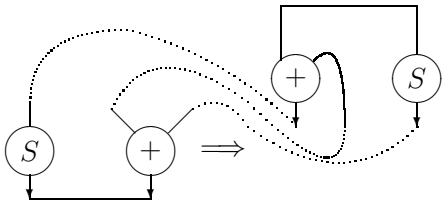
When replacing equals for equals in this notation, we observe that the right-hand side of the rule is always empty. In this case we will omit the '$\Longrightarrow$' symbol. We also note that all rules can be written in a form $\alpha(..) = \beta(...) \Longrightarrow N$, and as such we replace the '=' by '><' so that we can distinguish an occurrence of a rule from an occurrence of an active pair. Thus, we can write the rules above as:

$$\mathsf{Z} >< \mathsf{Add}(x,x)$$

$$\mathsf{S}(\mathsf{Add}(b,y)) >< \mathsf{Add}(\mathsf{S}(b),y)$$

This notation offers some flexibility as to how to write the rules. To understand the relationship between these last two sets of rules and the diagrams we can re-draw the diagrams as follows:





where the dashed lines indicate how we have substituted equals for equals to obtain the last set of rules. These diagrams are nothing more than reorganised versions of the diagrams given previously, where we have drawn all agents in a common orientation (with the principal ports always in the same direction).

This notation therefore allows sufficient flexibility so that we can either write nets in a compact way (without any net on the right of the '$\Longrightarrow$') or in a more intuitive way where the right-hand side of the net is written out in full to the right of the arrow.

Syntax choice

Of the two alternative notations that we have used above, the first language that we considered uses a notation developed by Gay [8], and also related in [4] to Combinatory Reduction Systems (CRSs) [13]. The representation of a net in this language is nothing more than a "flattening" of the net, replacing ports of agents by names, and representing edges by two occurrences of a name. This notation is quite heavy since it is bound up with names (variables), but is quite straightforward to relate to the graphical notation. This language has been used as a notation for interaction nets in several works (for instance [4,3]).

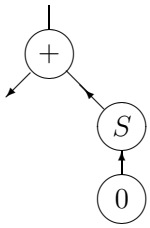
The second language that we have used is essentially that proposed by Lafont in [14] as a way of writing rules, and studied further in [5] to give a calculus of interaction nets. This notation is much lighter, but is difficult to grasp. This language however has gained quite a following, and indeed several implementations of interaction nets are based upon it.

The representation of nets that we use for our language will be the latter, but we will not insist that the nets are simplified by replacing equals for equals. In this way, one can write nets in a more natural way, combining the advantages of both notations.

3.2 Modularity

After choosing an appropriate syntax, one of the main issues is to add the ability to write nets in a modular way, and in particular to be able to name

nets and then compose them. For instance, suppose that the following net:



which we can write as $x = \mathsf{Add}(y, \mathsf{S}(\mathsf{Z}))$ in the syntax, was used often. Then we could introduce some syntax and define:

```
net Add1(x,y) => x=Add(y,S(Z))
```

This defines a net `Add1` which corresponds to a net which adds one to an argument. This net can be freely used anywhere where we would normally need a net (for instance in the definition of a rule). In particular, it can be used as an agent for the purpose of net construction. The ability to name nets can also be found elsewhere: Bechet introduced Abbreviations [2], and Gay introduced pseudo agents [8]. We also remark that a notion of combining nets, as a meta-operation, has been studied in [7].

The ability to name nets is essential so that a programmer can reuse the same net several times in a simple way. However, we also want to introduce a notion of encapsulation, which is a standard feature of modern programming languages. We propose a system of modules to allow the definition of parts of the system, which can then be combined. We therefore introduce the following syntax:

```
module Example {
 agents ...
 rules ...
 nets ...
}
```

which groups the definition of agents, rules and nets associated with these agents. Systems of interaction should then be able to import such modules, and in particular incorporate some form of qualified naming.

3.3 Data types

A very natural extension to interaction nets is the inclusion of primitive data types. Using the λ-calculus as an example, we can see this extension as adding delta rules, or building a very simple language such as PCF [24]. Thus, we should not have to build numbers as shown in the example above, but rather

have all the flexibility to assume such data, and also basic functions over them.

We propose to incorporate this by allowing agents to "hold" a value: `N{1}` is an agent that holds the integer 1. We then need to allow rules to modify these values. There is a choice here of either offering built-in functions, or providing a language in which these externally defined programs can be written. Here we simply restrict to a fixed language of arithmetic expressions. A discussion on different ways of doing this can be found in [6].

3.4 Side effects

Interaction nets are a pure calculus, and consequently if we want to allow side effects, input/output, file access, etc. then we must provide a suitable extension. We propose one such way by allowing an interaction rule to cause such effects in the present paper.

This completes our overview of what language features that would be required to offer a programming language for interaction nets. In the following sections we investigate these issues in more detail.

4 Syntax

In this section we give a syntax for interaction nets, and then give some indications how we expect this to be sugared up so that it can be used as a programming language. Following from the motivation section, we will write agents as $y = \alpha(x_1, \ldots, x_n)$, nets as comma separated lists of agents, and rules as the way a pair of agents can be replaced by a net. From now on, we try not to draw diagrams, but to use this notation.

Specification of agents

The two most basic facts about an agent are its name and arity. The arity of an agent is a very basic form of typing, and as such will be included in the declaration of agents in a program. We will therefore introduce agents together with a type. Using the example from the previous section, we have agent declarations as:

```
Z:0;
S:1;
Add, Mult:2;
```

Although there are many other aspects of agents that we could introduce here, we prefer to keep things simple, and constrain agents to be defined as above. We adopt a syntactical convention, which is not enforced by the compiler, that all agents begin with an upper case letter.

Specification of nets

Next we need a way of representing nets, for which we simply write the agents as a comma separated list, and use variable names to give the connectivity: if a variable occurs once in a term then it is a free edge, if it occurs twice then it is bound, if it occurs more than twice, then it is a syntax error. There are two special cases of a net: an empty net, which we shall denote by EMPTY, and an edge which we shall denote by $x = y$, where x and y are variable names.

All nets are written in a form where they are named, and thus we write:

```
net AddNet(r) => Add(r,S(Z))=S(Z)
```

for the net which corresponds to $S(0) + S(0)$.

Specification of rules

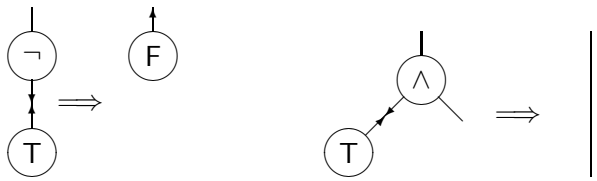
Finally, we need to express rules of an interaction system. From the diagram representing rules, we can write:

$$\alpha(x_1, \ldots, x_n) >< \beta(y_1, \ldots, y_m) \Longrightarrow N$$

where N is a net built from a comma separated list of agents. A rule must be closed in that all variable names should occur exactly twice. As we have seen above, we can perform some substitutions to simplify rules, but we are not obliged to. When we write rules, the order of α and β is not important. If the net N is empty, then we will just write:

$$\alpha(x_1, \ldots, x_n) >< \beta(y_1, \ldots, y_m)$$

We demonstrate this with a concrete example. Below are two rules which could be used as part of a system of interaction nets to represent Boolean expressions, together with negation and conjunction operators. The graphical form of example rules of this system are:



and the textual version would be either:

```
Not(x) >< True => x=False
And(x,y) >< True => x=y
```

or:

```
Not(False) >< True
And(x,x) >< True
```

An example net in this small system could be:

```
net Example(r) => True=And(And(r,True),True)
```

We leave the graphical representation of this net as a straightforward exercise to the reader.

This example shows that the syntax is a more compact representation than the graphical rules. However, the graphical rules are more intuitive, as are the nets. The point is that this syntax is still quite far from a programming language. With the introduction of naming of nets, we have also introduced the first keyword of the language "`net`".

The point of naming nets is to allow them to be combined. A typical scenario could be:

```
net N(x,y,z) => ...
net M(x,y,z) => ...
net K(x,y) =>  N(a,b,x),M(b,a,y)
```

Here we define nets `N` and `M`, which have free edges. The net `K` simply takes these nets and connects them together at the appropriate points. This is a very simple operation, and of course one that is the most important building block of the language.

Putting all these ideas together, we informally suggest the syntax of pure interaction nets:

- Definition of agents: each agent, together with some basic type information.
- Definition of rules: rules for pairs of agents. We stipulate that there is at most one rule for each pair of agents.
- Definition of nets: comma separated list of agents, where all the variables occur twice.

5 Modules

Modern programming languages do not expect a program to be written in a monolithic way (i.e. a list of agents, rules and nets in the case of interaction

nets, or as a single λ-term in the λ-calculus). We would expect some structure, and in particular, notions of modules and encapsulation. Here we want to bring out two features:

(i) Decompose structure of an interaction system into basic building blocks.

(ii) Introduce a mechanism for code reuse.

We begin by introducing some syntax directly, using the Boolean expressions as a running example:

```
module Boolean {
  True, False: 0;
  Neg(False) >< True
  Neg(True) >< False
}
module Erase {
   Epsilon: 0;
   Epsilon >< Epsilon
}
```

Now, suppose that we want to build a system which uses Booleans and also allows erasing:

```
module BoolErase {
  import Boolean, Erase;
  True >< Epsilon
  False >< Epsilon
  Neg(b) >< Epsilon => b=Epsilon
}
```

In this latter module we have `import`ed all the agents and rules from two other modules, and added a number of rules for the combined system.

We suggest also allowing selective importation of agents and rules, and also to allow the renaming of agents. This provides a mechanism for code reuse. A couple of examples are:

```
import True, False from Boolean;
import Neg as Not from Boolean;
```

The first line will import the agents from the module `Boolean`, and the second line will import the agent `Neg` from `Boolean` but will rename it (and its rules) to `Not`.

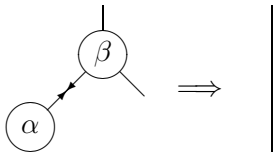
There are many other features that we could investigate here, such as qualified names for agents, etc., however, we will restrict the ideas here to the above.

Templates

Another use of the module system is to allow the definition of templates. Many interaction rules have the same shape, but differ only by the name of the agents. For example, consider the following three rules:

$$\begin{aligned} \mathsf{Add}(x,x) &\gg\!\ll \mathsf{Z} \\ \mathsf{Append}(x,x) &\gg\!\ll \mathsf{Nil} \\ \mathsf{And}(x,x) &\gg\!\ll \mathsf{True} \end{aligned}$$

which are of identical shape, but with different names of agents. As a diagram, all thee rules are:



where the pair (α, β) is one of: (Z, Add), (Nil, Append), or (True, And). The module system above allows a template to be defined once, and then imported into different systems:

```
module TemplateExample {
  A:2;
  B:1;
  Eps:0;
  A(x,x) >< B
  Eps >< B
  Eps >< A(Eps,Eps)
}
```

Now, we can write:

```
import A,B as Add,Z from TemplateExample;
```

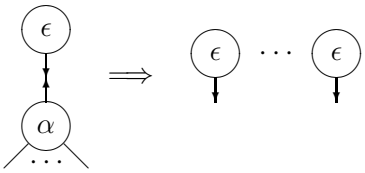
which will create a system with `Add` and `Z` as required.

Finally, we suggest an additional general form of template. `Epsilon` erases everything it interacts with, so we should define it just once, and then import it to everything that needs it. The rules can be generated automatically using the following, where "?" should be interpreted as any (all) agents of the given arity.

```
  module Erase {
    Eps : 1;
    Eps >< ?
```

```
  Eps >< ?(Eps)
  Eps >< ?(Eps,Eps)
  Eps >< ?(Eps,Eps,Eps)
}
```

Each of these rules defines the following scenario, for each arity of agent:



where α is any agent (in the same way as ? is any agent above).

This gives the interaction rules for erasing nets. There is scope for these definitions to be made generic by abstracting over the arity of "?", but we do not yet have a concrete proposal for such a facility.

6 Data-types

Interaction nets, like the pure λ-calculus, and also like first order term rewriting systems lack data types by default. We suggest extending interaction nets to include such data as part of the base system. We do this in a very naive way, by allowing agents to contain a value of a given base type.

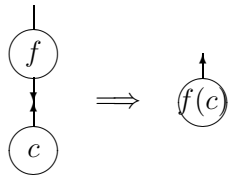
```
Num:0{int},
Str:0{string},
Flt:0{float};
```

Although it is possible to imagine more structure here, we prefer to limit the types to a fixed set.

Interaction rules can then access these values, and modify them. Again, we take quite a restrictive approach here, and only offer a simple language of expressions. Rules could then take on the following form:

```
Succ(a) >< Num{n} => a = Num{n+1}
```

where n on the left of a rule defines it as a pattern, and $n+1$ on the right can use the value of n. This idea can be seen as a diagram as follows:



where $f(c)$ is an externally defined reduction rule.

7 Input/Output

The final language features that we give for interaction nets is the inclusion of input and output, which require side-effecting reductions. For this we adopt the approach used by Gay [8], from which the example in this section is derived from.

To encode a mechanism for modelling input, we introduce the following agents:

```
InStream:0;
ReadNum:2;
```

and interaction rule:

```
ReadNum(y,x) >< InStream => x=Num{n},y=InStream
```

The auxiliary ports of `ReadNum` specify the destination of the received integer and the remainder of the input stream; the particular integer n associated with `Num{n}` would be determined at run-time. In a program with a sequence of `ReadNum` agents, the `InStream` is passed from one to the next so that the correct order of input values is maintained.

Output can be modelled by the following system:

```
OutStream:0;
PrintNum:2;
PrintNumAux:0{int};
```

and the interaction rules:

```
PrintNum(y,x) >< Num{n} => x=PrintNumAux(y){n};
PrintNumAux(y){n} >< OutStream => y=OutStream;
```

where it is understood that at run-time, the integer n associated with the `PrintNumAux` agent is displayed as output. It is necessary to introduce an agent `PrintNumAux` so that interaction rules remain binary.

A more structured approach to incorporating side-effects into interaction nets, which is under investigation, is based on the use of monads [12] in functional programming.

8 Conclusions

The goal of this paper was to make a first effort in the design of a programming language for interaction nets. Usability is one of our primary goals. There are

many extensions that can be made to the current proposal, and this is our current activity.

What we have presented is just the start of a development. We have identified a number of features that could, or rather should, be added to the current proposal: type systems, macro systems, graphical tools (graphical editor, or previewer).

An implementation of this proposal would provide a valuable tool to experiment with the language, and design in new features. This is one of our short-term goals.

References

[1] Asperti, A. and S. Guerrini, "The Optimal Implementation of Functional Programming Languages," Cambridge Tracts in Theoretical Computer Science **45**, Cambridge University Press, 1998.

[2] Bechet, D., *Partial evaluation of interaction nets*, in: M. Billaud, P. Castéran, M. M. Corsini, K. Musumbu and A. Rauzyand, editors, *Proceedings of the Second Workshop on Static Analysis WSA'92*, Bigre Journal **81-82**, 1992, pp. 331–338.

[3] Fernández, M. and I. Mackie, *Coinductive techniques for operational equivalence of interaction nets*, in: *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science (LICS'98)* (1998), pp. 321–332.

[4] Fernández, M. and I. Mackie, *Interaction nets and term rewriting systems*, Theoretical Computer Science **190** (1998), pp. 3–39.

[5] Fernández, M. and I. Mackie, *A calculus for interaction nets*, in: G. Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, Lecture Notes in Computer Science **1702** (1999), pp. 170–187.

[6] Fernández, M., I. Mackie and J. S. Pinto, *Combining interaction nets with externally defined programs*, in: *Electronic proceedings of the APPIA-GULP-PRODE Joint Conference on Declarative Programming*, 2001.

[7] Fernández, M., I. Mackie and J. S. Pinto, *A higher-order calculus for graph transformation*, Electronic Notes in Theoretical Computer Scince **72** (2002).

[8] Gay, S. J., *Interaction nets* (1991), Diploma in Computer Science Dissertation, University of Cambridge Computer Laboratory.

[9] Girard, J.-Y., *Linear Logic*, Theoretical Computer Science **50** (1987), pp. 1–102.

[10] Gonthier, G., M. Abadi and J.-J. Lévy, *The geometry of optimal lambda reduction*, in: *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)* (1992), pp. 15–26.

[11] Gonthier, G., M. Abadi and J.-J. Lévy, *Linear logic without boxes*, in: *Proceedings of the 7th IEEE Symposium on Logic in Computer Science (LICS'92)* (1992), pp. 223–234.

[12] Jones, S. P. and P. Wadler, *Imperative functional programming*, in: *20th ACM Symposium on Principles of Programming Languages (POPL'93)*, ACM Press, Charleston, 1993 pp. 71–84.

[13] Klop, J.-W., V. van Oostrom and F. van Raamsdonk, *Combinatory reduction systems, introduction and survey*, Theoretical Computer Science **121** (1993), pp. 279–308.

[14] Lafont, Y., *Interaction nets*, in: *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)* (1990), pp. 95–108.

[15] Lamping, J., *An algorithm for optimal lambda calculus reduction*, in: *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)* (1990), pp. 16–30.

[16] Lévy, J.-J., "Réductions Correctes et Optimales dans le Lambda-Calcul," Thèse d'état, Université Paris VII (1978).

[17] Lippi, S., *in^2 : A graphical interpreter for interaction nets*, in: S. Tison, editor, *Rewriting Techniques and Applications (RTA'02)*, Lecture Notes in Computer Science **2378** (2002), pp. 380–386.

[18] Mackie, I., *Static analysis of interaction nets for distributed implementations*, in: P. van Hentenryck, editor, *Proceedings of the 4th International Static Analysis Symposium (SAS'97)*, Lecture Notes in Computer Science **1302** (1997), pp. 217–231.

[19] Mackie, I., *Efficient λ-evaluation with interaction nets*, in: V. van Oostrom, editor, *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA'04)*, Lecture Notes in Computer Science **3091** (2004), pp. 155–169.

[20] Milner, R., M. Tofte, R. Harper and D. MacQueen, "The Definition of Standard ML (Revised)," MIT Press, 1997.

[21] Peyton Jones, S., editor, "Haskell 98 Language and Libraries," Cambridge University Press, 2003.

[22] Pierce, B. C. and D. N. Turner, *Pict: A programming language based on the pi-calculus*, Technical Report 476, Indiana (1997).

[23] Pinto, J. S., "Parallel Implementation with Linear Logic," Ph.D. thesis, École Polytechnique (2001).

[24] Plotkin, G., *LCF considered as a programming language*, Theoretical Computer Science **5** (1977), pp. 223–256.